

Finer-grained Locking in Concurrent Dynamic Planar Convex Hulls

K. Alex Mills
UT Dallas – ANDES Lab
800 W. Campbell Rd.
Richardson, TX USA
k.alex.mills@utdallas.edu

James Smith
UT Dallas – ANDES Lab
800 W. Campbell Rd.
Richardson, TX USA
james@nullious.net

ABSTRACT

The convex hull of a planar point set is the smallest convex polygon containing each point in the set. The dynamic convex hull problem concerns efficiently maintaining the convex hull of a set of points subject to additions and removals. One algorithm for this problem uses two external balanced binary search trees (BSTs) [16]. We present the first concurrent solution for this problem, which uses a *single* BST that stores references to intermediate convex hull solutions at each node. We implement and evaluate two lock-based approaches: a) fine-grained locking, where each node of the tree is protected by a lock, and b) “finer-grained locking,” where each node contains a separate lock for each of the left and right chains. In our throughput experiments, we observe that finer-grained locking yields an 8-60% improvement over fine-grained locking, and a 38-61 \times improvement over coarse-grained locking and software transactional memory (STM). When applied to find the convex hull of *static* point sets, our approach outperforms a parallel divide-and-conquer implementation by 2-4 \times using an equivalent number of threads.

Keywords

Dynamic planar convex hull, concurrent data structures, parallel convex hull.

1. INTRODUCTION

Historically, research on concurrent data structures has focused on general-purpose solutions which find common application (e.g. linked lists, skip-lists, binary search trees). We break from this tradition to focus on a specific problem: maintaining the convex hull of a *dynamic* planar point-set subject to insertions and deletions.

In the sequential setting, the planar convex hull problem is well-studied. There exist well-known algorithms which match the lower-bound of $\Omega(n \log n)$ [10, 1] and output-sensitive algorithms which run in $O(n \log h)$ time [12, 5], where h is the number of points on the hull.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2017 Copyright held by the owner/author(s).

ACM ISBN ... \$0.00

DOI:

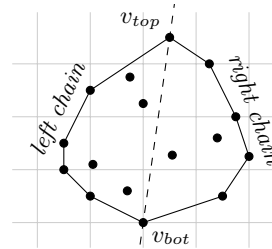
In the dynamic setting [17], Overmars and van Leeuwen presented an algorithm in which inserts and deletes can be achieved in $O(\log^2 n)$ time [16]. Later algorithms improved on this foundation, [6, 4] achieving $O(\log n)$ time for both inserts and deletes [4], but they are primarily of theoretical interest. The dynamic variant has various applications [3, 18, 7], including k-nearest neighbor search [9], least-squares clustering [2], and sketching geometric points [11].

2. DYNAMIC CONVEX HULLS

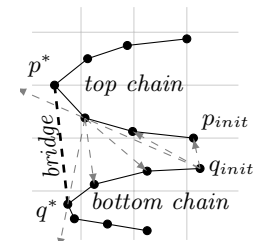
Throughout computational geometry it is standard practice to require that point sets are specified in *general position* w.l.o.g. [8]. In this paper and our experiments, general position means that a) no three points are co-linear, and b) no two points share a horizontal line.

A *convex polygon* is a simple polygon in which every internal angle is acute. The *convex hull* of a planar point set $S \subset \mathbb{R}^2$, denoted $\text{conv}(S)$, is the smallest convex polygon which contains all of the points in S . We use variables n and h to denote $n = |S|$ and $h = |\text{conv}(S)|$. In memory, we store convex hulls as a pair of linked lists, called the right and left chains. Each chain is a linked list of vertices v_1, v_2, \dots, v_h stored in *clockwise (c.w.)* order. Vertices adjacent in the list are joined by an edge; the edge from v_h to v_1 is implicitly included. If v_{top} and v_{bot} are the top and bottom-most points of S , then the *right chain* is the list of vertices in $\text{conv}(S)$ that lie on or to the right of the line from v_{top} to v_{bot} ; the *left chain* is the list of vertices which lie on or to the left of this line (Figure 1a).

Formally, the *dynamic (planar) convex hull problem* is the specification of an abstract data type which maintains a finite set of points $S \subset \mathbb{R}^2$ and provides the following three operations, 1) $\text{INSERT}(p)$: add point $p \in \mathbb{R}^2$ to S ,



(a) A convex hull



(b) Bridge-finding example (only left chains are shown).

Figure 1: Terminology and bridge-finding example.

- 2) `DELETE(p)`: remove point $p \in \mathbb{R}^2$ from S (if present),
- 3) `GETHULL()`: return the convex hull of S .

Divide-and-conquer for Convex Hulls.

We modify the approach of [16], which is based on the following divide-and-conquer algorithm. Given a *static* set of points $S \subset \mathbb{R}^2$, we compute the left and right chains of the convex hull separately. For concision, we describe only the left chain w.l.o.g.; the right chain is symmetric.

The divide step recursively splits S into two equal halves, S_{top} and S_{bot} , by the horizontal line formed by the median y -coordinate of S . The recursion terminates when two or fewer points $\{p_1, p_2\}$ remain. The *left* chain of $\{p_1, p_2\}$ is the points sorted in *increasing* order of y -coordinate (sorting ensures the c.w. ordering of the left chain). The conquer step combines the left chains of S_{top} and S_{bot} to form the left chain of their union $S_{top} \cup S_{bot}$. To combine the left chains of S_{top} and S_{bot} , we find and add the *bridge*, the unique line segment tangent to both chains. In Figure 1b, the bridge is the segment between p^* and q^* . The vertices p^* and q^* define the bridge and can be found by walking the top chain in clockwise order and walking the bottom chain in counter-clockwise order (Algorithm 1).

Algorithm 1: Bridge-finding algorithm

Input: a top and bottom chain separated by horiz. line
 $p \leftarrow$ bottom-most point of top chain (p_{init});
 $q \leftarrow$ top-most point of bottom chain (q_{init});
repeat
 while (p, q) not tangent to top chain **do**
 $p \leftarrow$ next point on top chain in c.w. order;
 while (p, q) not tangent to bottom chain **do**
 $q \leftarrow$ next point on bottom chain in c.c.w. order;
until (p, q) tangent to top and bottom chain;
return (p, q) ;

The dashed lines in Figure 1b depict the tangents tested in the while loops of Algorithm 1. Once the bridge is found, the lists representing the top and bottom chains are *split* at the points p^* and q^* . The (left) chain of $S_{top} \cup S_{bot}$ is formed by concatenating the part of the top chain above and including p^* with the bottom chain below and including q^* .

The parallel convex hull strategy we test against uses this algorithm. To avoid expensive median-finding, we sort by y -coordinate using `Arrays.parallelSort()` [14]. Each recursive call on a subproblem with more than 2000 points is submitted to a `ForkJoinPool` [15]. Recursive calls on fewer than 2000 points are solved sequentially. Cutting off parallelism after 2000 points yields good performance for our experimental workload in particular.

Dynamic Convex Hulls using BSTs.

Following [16], we store the solutions to divide-and-conquer subproblems at the internal nodes of a *HullTree*, an external BST in which each (internal) node has either zero or two children. Leaf nodes store the points of S sorted in order by y -coordinate. We refer to the two children of an internal node as the *top* and *bottom* children, which store both chains of the top and bottom subproblems respectively. Each node u stores a reference to the left and right chains of all the points stored at the leaf nodes below u . Just as in BSTs, we store the minimum key from the top subtree to aid in

routing search operations.

Inserting or deleting point p from the *HullTree* occurs as usual except it is immediately followed by a leaf-to-root pass back up the tree to recompute the chains. During this pass, if u has children c_1 and c_2 , we execute the conquer step on both chains stored at c_1 and c_2 , and *replace* the chains stored at node u with *copies* of the new chains which result. The leaf-to-root pass continues at u 's parent. This procedure propagates all needed changes to the root node; it recomputes only the subproblems whose solutions could change.

3. FINER-GRAINED LOCKING

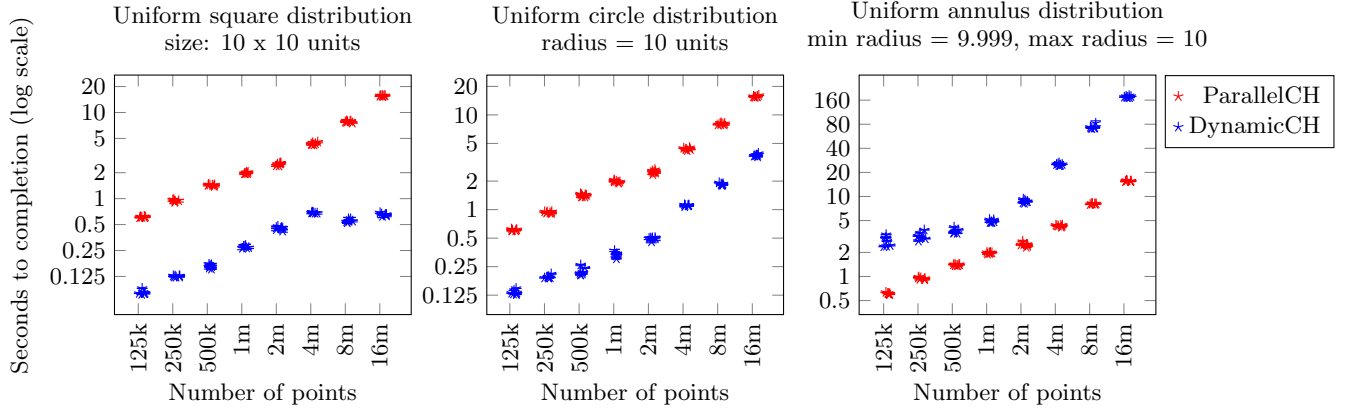
To manage concurrency in the “finer-grained locking” strategy, we store a *leftLock*, a *rightLock*, and an *isDeleted* flag at each node in addition to the child, parent, and chain references mentioned in Section 2. *leftLock* and *rightLock* are used to protect the left and right chains, respectively. We also use *leftLock* to protect access to a node's parent and child pointers. The fine-grained variant uses only one lock to protect access to all of a node's data members.

In both strategies, reads only need to access the convex hull stored at the root. We describe the procedure for writes as follows. Writes involving point p begin with a search down the *HullTree* to find the leaf node closest to p . During this search, locks are not acquired. Let u be the leaf node found by a search for point p .

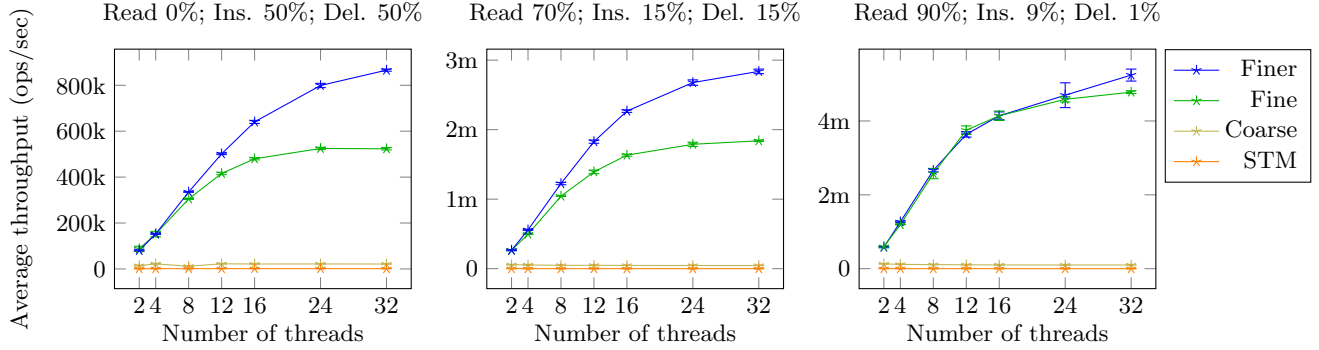
In case of an insert, we lock $u.leftLock$. Next, we validate to ensure a) that u is still a leaf node, and b) that $u.isDeleted$ is false. If validation fails, the insert operation restarts from the root. Otherwise, the leaf receives two newly allocated child nodes, one which contains p and another which contains the point stored at u . Then, both chains of u are updated so that they are the left/right chains of the points contained at u 's children. Finally, $u.leftLock$ is released and `merge(u)` is invoked (Algorithm 2).

In case of a delete, we acquire a three-node window of locks in this sequence: 1) we lock $u.leftLock$, 2) we *attempt* to lock the *leftLock* of u 's sibling, denoted u_{sib} , 3) we lock the *leftLock* of u 's parent, denoted u_{par} . Thus locks are acquired bottom-up. If the attempt to lock u_{sib} fails, locks are released, and the operation is retried from the root. Once all locks are acquired, the window is validated to ensure 1) that none of u , u_{sib} , or u_{par} have their *isDeleted* flag set to true, and 2) that u_{sib} and u are still children of u_{par} . If validation fails, the operation is retried from the root. Finally, the delete target is effectively removed from the tree via the following sequence of updates 1) we replace u_{par} 's left/right chains by the left/right chain of u_{sib} , 2) we redirect u_{par} 's top/bottom child pointers to point to the top/bottom children of u_{sib} , denoted c_1 and c_2 , 3) we redirect the parent pointers of c_1 and c_2 to refer to u_{par} , and 4) we set u and u_{sib} 's *isDeleted* flag to true. Finally, all locks are released and `merge(u_{par})` is invoked (see Algorithm 2).

The merge operation (Algorithm 2) performs “finer-grained hand-over-hand locking” up the tree, allowing updates to the left and right chains to occur concurrently while ensuring no pair of merge operations cross on their way to the root. Since *leftLock* is acquired first, if a merge collides with a write, the merge cannot proceed until the write has released its locks. Since writes only acquire leftLocks, it is possible for a write operation to overlap a merge while the merge is updating the right chain. Since writes are immediately



(a) Six instances of each size were averaged over 10 runs. Random noise added to x -coordinates for visual clarity.



(b) Each point averages six, 15-second runs. Measurements taken after a 2-second warmup period. 95% confidence intervals are shown.

Figure 2: Experimental results.

followed by a merge, this does not yield inconsistencies. We implement a minor optimization in which merges stop early once the chains stop changing as a result of the updates.

Algorithm 2: Merge operation.

Procedure `merge(node)`

```

prev ← null;
while node ≠ null do
  node.leftLock.lock();
  if prev ≠ null then prev.rightLock.unlock();
  { ... update left chain ... }
  node.rightLock.lock();
  node.leftLock.unlock();
  { ... update right chain ... }
  prev ← node; node ← node.parent;
if prev ≠ null then prev.rightLock.unlock();

```

Safety and Consistency Properties.

Our approach is easily seen to be deadlock-free, since 1) locks are always acquired bottom-up, and 2) *leftLock* is always acquired before *rightLock*. In finer-grained locking, reads may find the root node in an inconsistent state when the top/bottommost points of the hull change due to concurrent writes. This condition may be tested for by ensuring that the top/bottommost points in the left and right chains match. If they do not, the situation can be remedied by (a) retrying until a clean read occurs (linearizability), (b) taking the convex hull of the result in $O(h)$ time (quiescent consistency), or (c) returning a possibly inconsistent

view to the caller. Since inconsistent intermediate views are acceptable in *static* workloads, we use approach (c).

4. EXPERIMENTS AND CONCLUSIONS

Figure 2a depicts three experiments on static instances, each run using 24 threads. They were performed on a 64-bit 12-core AMD Opteron 6180 SE with 64 GB of RAM, (24 hyperthreads). “ParallelCH” is the parallel divide-and-conquer implementation previously mentioned. “DynamicCH” tests each point to see if it lies in the convex hull of the finer-grained concurrent data structure. If the point is not already in the convex hull, it is inserted into the data structure, otherwise it is ignored. All implementations are in Java. In the left and middle plots, instances are formed by drawing points uniformly at random from a box and a circle respectively. The right plot is a stress test in which points are sampled from an extremely thin annulus (area: 0.062 units²). This scenario is *far from a typical case*: the sampled points are extremely close to the boundary in a *tiny* region measuring only 1/5000× the area of the circle used in the middle plot. This distribution forces the dynamic data structure to manage many large chains. Yet even in this case, using finer-grained locking yields a throughput speedup of between 8–60%. In Figure 2b three workloads *using the same annulus distribution* are shown. These experiments were performed on a machine using two 64-bit 16-core Intel Xeon E5-4650 processors with 64GB of RAM (32 threads). DeuceSTM v1.3 was used in the STM variant [13].

In future work, we propose investigating balanced Hull-Trees and the space-saving techniques described in [16].

5. ACKNOWLEDGEMENTS

We are grateful to Neeraj Mittal, Shreyas Gokhale, and Kenneth Platz for insightful conversations and feedback on the design and experimental analysis phases of this project. We would also like to thank Shreyas Gokhale, Joel Long, and Gregory van Buskirk for their constructive feedback on drafts.

6. REFERENCES

- [1] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inf. Process. Lett.*, 9(5):216–219, 1979.
- [2] F. Aurenhammer, F. Hoffmann, and B. Aronov. Minkowski-type theorems and least-squares clustering. *Algorithmica*, 20(1):61–76, 1998.
- [3] S. Bespamyatnikh, D. Kirkpatrick, and J. Snoeyink. Generalizing ham sandwich cuts to equitable subdivisions. *Discrete & Computational Geometry*, 24(4):605–622, 2000.
- [4] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. 43rd Symp. Foundations of Comput. Sci.*, pages 617–626, 2002.
- [5] T. M. Chan. *Output-sensitive Construction of Convex Hulls*. PhD thesis, Univ. British Columbia, Vancouver, BC, Canada, 1995.
- [6] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48(1):1–12, Jan. 2001.
- [7] B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *J. ACM*, 34(1):1–27, Jan. 1987.
- [8] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer Publishing Co., Inc., 3rd edition, 2010.
- [9] H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, 15(1):271–284, 1986.
- [10] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1(4):132–133, 1972.
- [11] J. Hershberger, N. Shrivastava, and S. Suri. Summarizing spatial data streams using clusterhulls. *ACM J. Experimental Algorithmics*, 13, 2008.
- [12] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299, 1986.
- [13] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. <http://mcg.cs.tau.ac.il/papers/multiprog10-deuce.pdf/>, 2010.
- [14] Oracle Corporation. Arrays. <https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>, 2014–2016.
- [15] Oracle Corporation. ForkJoinPool. <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html>, 2014–2016.
- [16] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.
- [17] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Commun. ACM*, 22(7):402–405, 1979.
- [18] A. Tamir. Improved complexity bounds for center location problems on networks by using dynamic data structures. *SIAM J. Discrete Math.*, 1(3):377–396, 1988.